9 Main Class

Every program must have a class `Main`. Furthermore, the `Main` class must have a method `main` that takes no formal parameters. The `main` method must be defined in class `Main` (not inherited from another class). A program is executed by evaluating `(new Main).main()`.

The remaining sections of this manual provide a more formal definition of Cool. There are four sections covering lexical structure (Section 10), grammar (Section 11), type rules (Section 12), and operational semantics (Section 13).

10 Lexical Structure

The lexical units of Cool are integers, type identifiers, object identifiers, special notation, strings, keywords, and white space.

10.1 Integers, Identifiers, and Special Notation

Integers are non-empty strings of digits 0-9. Identifiers are strings (other than keywords) consisting of letters, digits, and the underscore character. Type identifiers begin with a capital letter; object identifiers begin with a lower case letter. There are two other identifiers, **self** and **SELF_TYPE** that are treated specially by Cool but are not treated as keywords. The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are given in Figure 1.

10.2 Strings

Strings are enclosed in double quotes `"..."`. Within a string, a sequence '\c' denotes the character 'c', with the exception of the following:

\b	backspace
\t	tab
\n	newline
\f	formfeed

A non-escaped newline character may not appear in a string:

```
"This \
is OK"
"This is not
OK"
```

A string may not contain EOF. A string may not contain the null (character \0). Any other character may be included in a string. Strings cannot cross file boundaries.

10.3 Comments

There are two forms of comments in Cool. Any characters between two dashes "`--`" and the next newline (or EOF, if there is no next newline) are treated as comments. Comments may also be written by enclosing text in (∗ . . . ∗). The latter form of comment may be nested. Comments cannot cross file boundaries.

10.4 Keywords

The keywords of cool are: **class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true.** Except for the constants **true** and **false**, keywords are case insensitive. To conform to the rules for other objects, the first letter of **true** and **false** must be lowercase; the trailing letters may be upper or lower case.

10.5 White Space

White space consists of any sequence of the characters: blank (ascii 32), `\n` (newline, ascii 10), `\f` (form feed, ascii 12), `\r` (carriage return, ascii 13), `\t` (tab, ascii 9), `\v` (vertical tab, ascii 11).

11 Cool Syntax

Figure 1 provides a specification of Cool syntax. The specification is not in pure Backus-Naur Form (BNF); for convenience, we also use some regular expression notation. Specifically, A^* means zero or more A's in succession; A^+ means one or more A's. Items in square brackets [. . .] are optional. Double brackets $[\![\ ]\!]$ are not part of Cool; they are used in the grammar as a meta-symbol to show association of grammar symbols (e.g. $a[\![bc]\!]^+$ means a followed by one or more bc pairs).

11.1 Precedence

The precedence of infix binary and prefix unary operations, from highest to lowest, is given by the following table:

```
.
@
~
isvoid
* /
+ -
<=  <  =
not
<-
```

All binary operations are left-associative, with the exception of assignment, which is right-associative, and the three comparison operations, which do not associate.

12 Type Rules

This section formally defines the type rules of Cool. The type rules define the type of every Cool expression in a given context. The context is the *type environment*, which describes the type of every unbound identifier appearing in an expression. The type environment is described in Section 12.1. Section 12.2 gives the type rules.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \llbracket \mathit{class}; \rrbracket^{+} \\
\mathit{class} & ::= & \textbf{class}\ \text{TYPE}\ [\textbf{inherits}\ \text{TYPE}]\ \{\ \llbracket \mathit{feature}; \rrbracket^{*} \} \\
\mathit{feature} & ::= & \text{ID}(\ [\ \mathit{formal}\ \llbracket , \mathit{formal} \rrbracket^{*}\ ]\ ) : \text{TYPE}\ \{\ \mathit{expr}\ \} \\
 & | & \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ] \\
\mathit{formal} & ::= & \text{ID} : \text{TYPE} \\
\mathit{expr} & ::= & \text{ID}\ \texttt{<-}\ \mathit{expr} \\
 & | & \mathit{expr}[@\text{TYPE}].\text{ID}(\ [\ \mathit{expr}\ \llbracket , \mathit{expr} \rrbracket^{*}\ ]\ ) \\
 & | & \text{ID}(\ [\ \mathit{expr}\ \llbracket , \mathit{expr} \rrbracket^{*}\ ]\ ) \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{expr}\ \textbf{else}\ \mathit{expr}\ \textbf{fi} \\
 & | & \textbf{while}\ \mathit{expr}\ \textbf{loop}\ \mathit{expr}\ \textbf{pool} \\
 & | & \{\ \llbracket \mathit{expr}; \rrbracket^{+} \} \\
 & | & \textbf{let}\ \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ]\ \llbracket , \text{ID} : \text{TYPE}\ [\ \texttt{<-}\ \mathit{expr}\ ] \rrbracket^{*}\ \textbf{in}\ \mathit{expr} \\
 & | & \textbf{case}\ \mathit{expr}\ \textbf{of}\ \llbracket \text{ID} : \text{TYPE} => \mathit{expr}; \rrbracket^{+} \textbf{esac} \\
 & | & \textbf{new}\ \text{TYPE} \\
 & | & \textbf{isvoid}\ \mathit{expr} \\
 & | & \mathit{expr} + \mathit{expr} \\
 & | & \mathit{expr} - \mathit{expr} \\
 & | & \mathit{expr} * \mathit{expr} \\
 & | & \mathit{expr} / \mathit{expr} \\
 & | & \tilde{}\,\mathit{expr} \\
 & | & \mathit{expr} < \mathit{expr} \\
 & | & \mathit{expr} <= \mathit{expr} \\
 & | & \mathit{expr} = \mathit{expr} \\
 & | & \textbf{not}\ \mathit{expr} \\
 & | & (\mathit{expr}) \\
 & | & \text{ID} \\
 & | & \text{integer} \\
 & | & \text{string} \\
 & | & \textbf{true} \\
 & | & \textbf{false}
\end{array}
$$

Figure 1: Cool syntax.